Multi Robots Task Assignments Problems Solved by Evolutionary Algorithm

Chen Peng [a]

[a] *Mechanical and Aerospace Engineering Department, UC, Davis, US*
(penchen@ucdavis.edu,)

Abstract: This term paper mainly uses genetic algorithm to solve the multi robots task assignment(MRTA) problems in two situations. We assume that each robot can go directly from one point to another and the distance is just the euclidean distance between two points. One task only needs one robot in one assignment and all tasks need to be visited in one task execution. In the first situation, we have the same amounts of robots as the tasks (m vs m), while in the second condition, we have less robots than tasks. The objective function of both problems is to find the shortest summary distances of all the robots to go over all task points.

Keywords: MRTA, evolutionary algorithm, TSP, mTSP

1 INTRODUCTION

Multi-robot systems (MRS) are a group of robots that are designed aiming to perform some collective behavior.One of the most challenging problems of MRS is how to optimally assign a set of robots to a set of tasks in such a way that optimizes the overall system performance subject to a set of constraints.This problem is known as Multi-robot Task Allocation (MRTA) problem Alaa Khamis and Elmogy [2015]. In the first situation, we want to match each robot with one particular task so that the total distance is shortest. This is very similar to the classic problem of traveling salesman problem: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"wik [2016] In the second condition, we consider less robots to visit more tasks. This problem is very similar to mTSP problem: The multiple traveling salesman problem (MTSP) involves scheduling m $<$1 salesmen to visit a set of n $>$m nodes so that each node is visited exactly once.Sedighpour et al. [2012]

2 M ROBOTS M TASKS

In the first situation, we assume that the robots are just equal to the amount of tasks and we want to choose the optimal task allocations to make minimum the total distances of all the robots will go over. If we solve this problem by exhaustive permutation of all the robots (or tasks), it will be a NP hard problem and the complexity is O(n!). Therefore, we need to choose other methods to decrease the computation complexity to make the problem solvable in high dimension.

2.1 Evolutionary algorithm model

In the process of finding optimal result, we use the method of Hill-Climbing. This technique is related to gradient ascent, but it does not require you to know the strength of

the gradient or even its direction: you just iteratively test new candidate solutions in the region of your current candidate, and adopt the new ones if theyre better. This enables you to climb up the hill until you reach a local optimum.Luke [2013] At the same time, we also need to try the global random searching to avoid the local optimal. Here we use the method of simulated annealing to sometimes accept worse solutions with a certain probability and this probability decreases over time based on 'initial temperature' and 'cooling schedule'Johnson et al. [1989].

2.2 Small numbers trial

To test the convergence effect and speed of our algorithm, we use exhaustive way to obtain the global minimum to testify the convergence of our algorithm.

Num of pairs	Global minimum	Optimal searched
5	23.6015	23.6015
8	21.781	22.1582
10	37.531	37.531
12	66.469	66.73

We can also visualize the result and show the converging process in Figure 1.

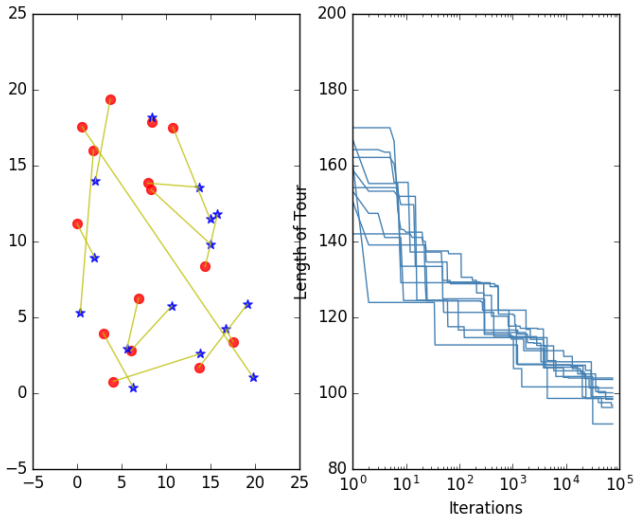

Figure 1. Visualization of task assignment and searching convergence

2.3 Large numbers solving

Photographs should only be used if essential to the clarity of the paper. If used they must be with clear contrast and highly glossed.

2.4 Complexity of algorithm

The best upper bound should be very similar to TSP problem. If we set the search range at [1,1], then we can use the equation from Steinerberger [2015]. The equation list below:

$$L = \beta\sqrt{n}$$

β is equal to 0.92

3 M ROBOTS N TASKS(M <N)

3.1 Robot initial alignment

We can make the initial alignment of the robots to make it as average distributed in the map as possible. In this way, each robot can more likely choose similar amount of tasks and the efficiency of the whole system can be more efficient.

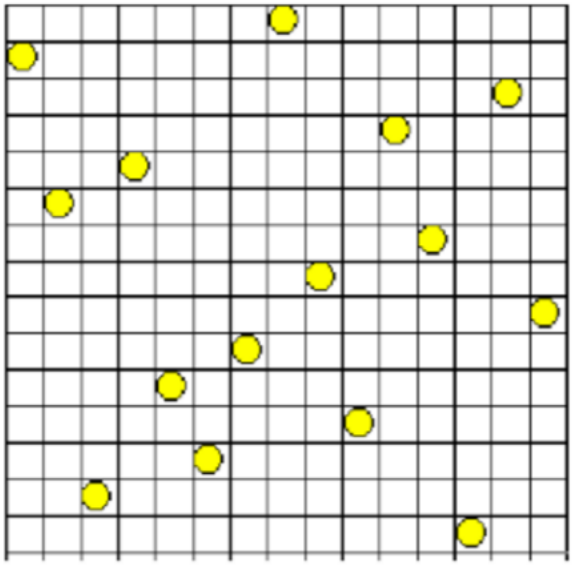

Figure 2. Visualization of task assignment and searching convergence

3.2 Local search

At first, we make each task to choose the robot which is nearest to it and then we can obtain the task assignments for each robot. This ”Local search” is to meet the requirement that each task is only need one robot to be visited.

3.3 MRTA solver

After each robot obtain their task allocation. We can use the TSP method to calculate the minimum distance of each robots will go over their tasks. The result can be visualized in

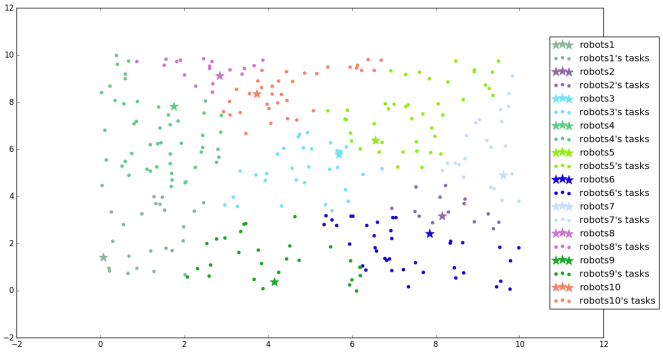

Figure 3. Visualization of task assignment and searching convergence

Figure 4.

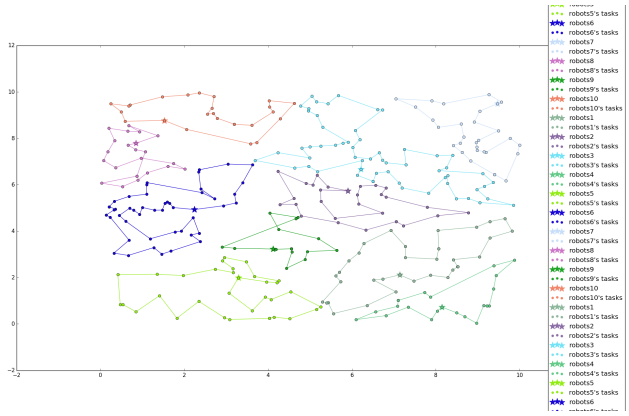

Figure 4. Visualization of task assignment and searching convergence

ACKNOWLEDGMENTS

This project is advised by Professor Jon Herman and most of the methods I used is from his courses **ECI 289i**. I very appreciate his great help and awesome course contents in and out of the class.

REFERENCES

Traveling sales man problem, Dec 2016. `https://en.wikipedia.org/wiki/Travelling_salesman_problem`.

Alaa Khamis, A. H. and A. Elmogy. Multi-robot task allocation: A review of the state-of-the-art. *Some Cooperative Robots and Sensor Networks 2015*, vol 7:31–51, 2015.

Johnson, D. S., C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; part i, graph partitioning. *Operations research*, 37(6):865–892, 1989.

Luke, S. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

Sedighpour, M., M. Yousefikhoshbakht, and N. Mahmoodi Darani. An effective genetic algorithm for solving the multiple traveling salesman problem. *Journal of Optimization in Industrial Engineering*, (8):73–79, 2012.

Steinerberger, S. New bounds for the traveling salesman constant. *Advances in Applied Probability*, 47(01):27–36, 2015.

4 APPENDICES

4.1 Code of m robots m tasks allocation

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

verbose = False
save = False

''' ##################################################
  Generate task points for a certain area
  range:(xrange, yrange), N is number of tasks points
'''##################################################
def get_random_tasks(search_range,N):
  x = np.random.random(N)*search_range[0]
  y = np.random.random(N)*search_range[1]
  xy = np.vstack((x,y))
  if verbose:
    print "Tasks points: ", xy
  if save:
    name = 'task_points-'+str(N)+'.txt'
    np.savetxt(name,xy)
  return xy
''' ##################################################
      Distribute the robots in the task range
'''##################################################
def lhs(N,D):
  grid = np.linspace(0,1,N+1)
  result = np.random.uniform(low=grid[:-1], high=grid[1:], size=(D,N))
  for c in result:
    np.random.shuffle(c)
  return result

def assign_robots(search_range,N):
  x = lhs(N,1)*search_range[0]
  y = lhs(N,1)*search_range[1]
  xy = np.vstack((x,y))
  if verbose:
    print "Robots points: ", xy
  if save:
    name = 'task_points-'+str(N)+'.txt'
    np.savetxt(name,xy)
```

```
  return xy

''' ################################################
      Local search: (O(n*logn*m)+O(n**2))
      Tasks asks help for local robots
      It can only calls the robots nearsets to them
'''################################################
def raw_allocation(robots,tasks,search_range):
  num_robots = robots.shape[1]
  num_tasks  = tasks.shape[1]
  dist_c = np.zeros((num_robots,num_tasks)) # distance matrix

  for i in range(num_robots):
    for j in range(num_tasks):
      dist_c[i,j] = np.sqrt(((robots[:,i]-tasks[:,j])**2).sum())

  if verbose:
    print dist_c.shape
  # Each task only need one nearest robot to help
  robot_chosen = np.argmin(dist_c,axis=0)

  if verbose:
    print "Each task choose robot:", robot_chosen

  # Task assignment of robots
  robots_tasks = dict()
  robots_tasks = {k: [] for k in range(num_robots)}
  # robots_tasks = robots_tasks.fromkeys(robot_list,[])
  for i,robot in enumerate(robot_chosen):
    robots_tasks[robot].append(i) # each elements means which number of tasks is ch

  if verbose:
    print "robot choose tasks:", robots_tasks
  allocation_show(robots,tasks,robots_tasks,search_range)

  return robots_tasks
''' ################################################
      Local TSP: (Go through robots+task points)
      Task execution of each robots
'''################################################
def distance(tour, xy):
  # index with list, and repeat the first city
  tour = np.append(tour, tour[0])
  d = np.diff(xy[tour], axis=0)
  return np.sqrt((d**2).sum(axis=1)).sum()

def OneRobotTask_solver(num_seeds, max_NFE, xy, search_range):
  # calculate the theoritical minimum of search length
  num_rtsk = xy.shape[0]
  search_length = min(search_range)
  L_star = np.sqrt(num_rtsk)*0.25*search_length

  # time mark of each calculation
  ft = np.zeros((num_seeds, max_NFE))
  exect = np.arange(num_rtsk)
```

```
  exectt = np.zeros((num_seeds,num_rtsk)).astype(int)

  # task execution
  for seed in range(num_seeds):
    np.random.seed(seed)
    # random initial tour
    exect = np.random.permutation(exect)
    bestf = 99999
    n = 0
    while n < max_NFE and bestf > L_star:
    # mutate the tour using two random cities
      trial_exect = np.copy(exect) # do not operate on original list
      a = np.random.randint(num_rtsk)
      b = np.random.randint(num_rtsk)
    # the entire route between two cities are reversed
      if a > b:
        a,b = b,a
        trial_exect[a:b+1] = trial_exect[a:b+1][::-1]

      trial_f = distance(trial_exect, xy)

      if trial_f < bestf:
        exect = trial_exect
        bestf = trial_f
      ft[seed,n] = bestf
      n += 1
      exectt[seed] = exect
  # print exect
  # print bestf
  best_ind = np.unravel_index(np.argmin(np.nonzero(ft)), ft.shape)
  bestf = ft[best_ind]
  bestexect = exectt[best_ind[0]]

  return bestexect,bestf,ft # best result of all seeds

''' ##################################################
    Global TSP: summary of all robots execution
    Task execution of all robots
'''##################################################
def MRTA_Solver(robots,tasks,num_seeds,max_NFE,search_range):
  robots_tasks = raw_allocation(robots,tasks,search_range) # robot task assignment

  # change task assignment to points coordinates
  MRTA_EXE = np.zeros(robots.shape[1])
  MRTA_tour = dict()
  MRTA_tour = {k: [] for k in range(robots.shape[1])}
  for i in range(robots.shape[1]):
    if verbose:
      print robots[:,i].shape
      print tasks[:,robots_tasks[i]].shape
    OneRobotTask = np.vstack((robots[:,i].T,tasks[:,robots_tasks[i]].T))
    # print OneRobotTask
  # find best tour and shortest distance for each robot
    besttour, bestf, ft = OneRobotTask_solver(num_seeds,max_NFE,OneRobotTask,search
  # Mark all the robots execution
```

```
    MRTA_tour[i] = OneRobotTask[besttour,:]
    MRTA_EXE[i] = bestf

    print "%d of %d robot executing tasks" %(i+1 , robots.shape[1])
    print "Task assignments", robots_tasks[i]
    print "Running distances", MRTA_EXE[i]

  # print MRTA_tour
  if save:
    filename = "MRTA_tour.npy"
    np.save(filename,MRTA_tour)
    print MRTA_tour

  return MRTA_tour, MRTA_EXE
''' ####################################################
      Visualizations of task assignment and execution
'''####################################################

def exect_map_show(MRTA_tour,robots,tasks,search_range):
  # plt.subplot(1,1,1)
  color_arr = color_produce(robots)
  for i in range(robots.shape[1]):
    MRTA_tour[i] = np.vstack((MRTA_tour[i], MRTA_tour[i][0])) # for plotting
    # print MRTA_tour[i]
    plt.plot(MRTA_tour[i][:,0], MRTA_tour[i][:,1], marker='o', color=color_arr[i,:]
  plt.show()

def color_produce(robots):
  np.random.seed(0)
  robot_num = robots.shape[1]
  color_arr = np.random.rand(robot_num,3)

  return color_arr
def allocation_show(robots,tasks,robots_tasks,search_range):
      # plt.scatter(robots[0,:], robots[1,:], marker='o', color="r",alpha=0.3)
  color_arr = color_produce(robots)
  plt.subplot(1,1,1)
  for i in range(robots.shape[1]):
    color_r = color_arr[i,:]
    r_label = 'robots'+str(i+1)
    rt_label = r_label+'\'s tasks'
    plt.scatter(robots[0][i], robots[1][i], marker='*', color=color_r,s=200,label=
    plt.scatter(tasks[:,robots_tasks[i]][0,:],tasks[:,robots_tasks[i]][1,:],marker=
  plt.legend(loc='center left',bbox_to_anchor=(0.9, 0.5))
  plt.show()

# NFE to different cities
num_robots = 10 # tasks equal to num of robots
num_tasks = 300
search_range = [10,10]
num_seeds = 10
max_NFE = 100000
robots = assign_robots(search_range,num_robots)
tasks = get_random_tasks(search_range,num_tasks)
```

```
robots_tasks = raw_allocation(robots,tasks,search_range)
allocation_show(robots,tasks,robots_tasks,search_range)
# MRTA_tour, MRTA_EXE = MRTA_Solver(robots,tasks,num_seeds,max_NFE,search_range)
# exect_map_show(MRTA_tour,robots,tasks, search_range)
```

4.2 Code of m robots execute n tasks

```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import itertools

verbose = False
save = False
exhaustive = True
# range (xrange, yrange), N is number of tasks points
def get_random_tasks(search_range, N):
  x = np.random.random(N)*search_range[0]
  y = np.random.random(N)*search_range[1]
  xy = np.vstack((x,y))
  if verbose:
    print "Tasks points: ", xy
  if save:
    name = 'task_points-'+str(N)+'.txt'
    np.savetxt(name,xy)
  return xy

def get_random_robots(search_range, N):
  x = np.random.random(N)*search_range[0]
  y = np.random.random(N)*search_range[1]
  xy = np.vstack((x,y))
  if verbose:
    print "Robots points: ", xy
  if save:
    name = 'task_points-'+ str(N) + '.txt'
    np.savetxt(name,xy)
  return xy

def distance(robots, tasks):
  d = robots - tasks
  return (np.sqrt((d**2).sum(axis=0))).sum()

def exhaustive_all(robots, tasks):
  # print tasks
  tasks_copy = np.copy(tasks)
  assignments = range(robots.shape[1])
  best_dist = distance(robots,tasks_copy)
  for assign in itertools.permutations(assignments):
    assign = np.array(list(assign))
    tasks_copy = tasks_copy[:,assign]
    # tasks_copy[0] = tasks_copy[1][assign]
    # tasks_copy[1] = tasks_copy[1][assign]
    dist = distance(robots,tasks_copy)
    if dist <= best_dist:
```

```
      best_dist = dist
      best_assign = assign
      # print best_dist
    tasks_copy = np.copy(tasks) # change to original one

  print "Best", best_dist
  print "Best task points", tasks_copy[:,best_assign]
  return best_dist

# def task_assignment_solver(num_seeds,max_NFE,tasks,robots, best_dist):
def task_assignment_solver(num_seeds,max_NFE,tasks,robots,exhaustive):
  if exhaustive:
    best_dist = exhaustive_all(robots,tasks)
  else:
    best_dist = 0
  # ft = []
  num_tasks = tasks.shape[1]
  ass = np.arange(num_tasks)
  assignt = np.zeros((num_seeds,num_tasks)).astype(int)
  # assignt = dict()
  # assignt = {k: [] for k in range(num_seeds)}
  max_NFE *= num_tasks
  ft = np.zeros((num_seeds, max_NFE))
  T0 = 2 # initial temperature
  alpha = 0.95 # cooling parameter
  for seed in np.arange(num_seeds):
    np.random.seed(seed)
    # random initial tour
    best_assign = np.random.permutation(ass)
    # assignment = ass
    initial_order = best_assign
    tasks_copy = np.copy(tasks)
    tasks_copy = tasks_copy[:,best_assign]
    bestf = distance(robots,tasks_copy)
    i = 0
    ft[seed,i] = bestf
    # assignt[seed].append(assignment)
    T=T0
    # for i in np.arange(max_NFE):
    while i < max_NFE-1 and bestf > best_dist*1.005:
  # mutate the tour using two random cities
      trial_assignment = np.copy(best_assign) # do not operate on original list
      # assignt = assignt[seed].append(trial_assignment)
      # a,b = np.random.random_integers(num_tasks,size=(2,))
      a = np.random.randint(num_tasks)
      b = np.random.randint(num_tasks)

    # the entire route between two cities are reversed
      if a > b:
        a,b = b,a
      trial_assignment[a:b+1] = trial_assignment[a:b+1][::-1]
    # if np.abs(bestf - best_dist) > 0.05:
    #   trial_assignment[a:b+1] = trial_assignment[a:b+1][::-1]

      tasks_copy = tasks_copy[:,trial_assignment]
```

```
        trial_f = distance(robots, tasks_copy)

        r = np.random.rand()
        if T > 10**-3: # protect division by zero
          P = np.min([1.0, np.exp((bestf - trial_f)/T)])
        # print P
        else:
          P = 0

        if trial_f <= bestf or r < P:
          best_assign = trial_assignment
          best_tasks = np.copy(tasks_copy)
          bestf = trial_f

        ft[seed,i] = bestf
        # print assignment
        # print bestf
        # ft[seed].append(bestf)
        i += 1
      # assignt[seed] = assignment
        T = T0*alpha**i
        # print a, b, seed, "seed", i, "trail", "r",r, "P", P, assignment, bestf

      assignt[seed] = best_assign
      # best_ind = np.unravel_index(np.argmin(ft[seed]), ft.shape)
      # assignment = assignt[best_ind]
      print "seed", seed, "best value", bestf
      # print "best tasks points list ", best_tasks
    # ft = np.array(ft)
    ft[ft==0] = np.inf
    # best_ind = np.unravel_index(np.argmin(np.nonzero(ft)), ft.shape)
    best_ind = np.unravel_index(np.argmin(ft), ft.shape)
    bestf = ft[best_ind]
    print best_ind
    bestassign = assignt[best_ind[0]]
    return best_tasks, bestf, ft, best_dist # best result of all seeds

def assignment_show(robots,tasks,ft):
    plt.subplot(1,2,1)
    plt.scatter(robots[0,:], robots[1,:], marker='o', color="r",alpha=0.8, s=50)
    plt.scatter(tasks[0,:],  tasks[1,:],  marker='*', color="b",alpha=0.8, s=50)
    # assignt = range(tasks.shape[1])
    for i, assign in enumerate(range(tasks.shape[1])):
      plt.plot([robots[0][i],tasks[0][assign]],[robots[1][i],tasks[1][assign]], colo
    # plt.plot(robots[:,0], robots[:,1])
    # plt.plot(tasks[:,0],tasks[:,1], marker='*')
    plt.subplot(1,2,2)
    # plt.semilogx(ft.T, color='steelblue', linewidth=1)
    plt.semilogx(ft.T, color='steelblue', linewidth=1)
    plt.xlabel('Iterations')
    plt.ylabel('Length of Tour')
    plt.show()

# NFE to different cities
num = 10 # tasks equal to num of robots
```

```
search_range = [20,20]
num_seeds = 10
max_NFE = 5000

np.random.seed(1)
robots =  get_random_robots(search_range,num)
tasks  =  get_random_tasks(search_range,num)

# best_dist, best_assign_theory = exhaustive_all(robots, tasks)

if exhaustive:
  best_tasks, bestf, ft = task_assignment_solver(num_seeds,max_NFE,tasks,robots,ex
else:
  best_tasks, bestf, ft, best_dist = task_assignment_solver(num_seeds,max_NFE,tasks

print "The best value found is", bestf
# print "The best assginment found is", bestassign
if exhaustive:
  print "The theory best value is", best_dist
# print "The theory best assginment is", best_assign_theory
# tasks_show = np.copy(tasks)[:,initial_order]
# tasks_show = tasks_show[:,initial_order]
assignment_show(robots,best_tasks,ft)
```